Рассматривается реализация файловой системы, монтируемой на блочное устройство, в операционных системах семейства Unix/Linux. Версия ядра: 5.8

**Форматирование блочного устройства.**

Чтобы адресовать данные на блочном устройстве системными вызовами VFS операционной системы, необходимо предварительно форматировать блочное устройство, то есть записать на него структуру суперблок и таблицу файловых дескрипторов.

Заметим, что структура суперблок и файловые дескрипторы, расположенные на диске, отличаются от соответствующих структур пространства ядра. Однако, структура суперблок пространства ядра ставится в соответствие структуре суперблок на диске. Аналогично, файловый дескриптор пространства ядра ставится в соответствие файловому дескриптору, расположенному на диске, при выполнении действий файловой системы над данными.

Структура суперблок, расположенная на диске:

typedef struct myrwfs\_super\_block

{

byte4\_t magic\_number; /\* Magic number - идентифицировать файловую систему \*/

byte4\_t block\_size\_bytes; /\* размер блока в байтах \*/

byte4\_t bdev\_size\_blocks; /\* размер устройства в блоках \*/

byte4\_t entry\_size\_bytes; /\* размер дескриптора файла в байтах \*/

byte4\_t entry\_table\_size\_blocks; /\* размер таблицы дескрипторов в блоках \*/

byte4\_t entry\_table\_block\_start; /\* смещение к началу таблицы дескрипторов в блоках - она расположена после суперблока на устройстве \*/

byte4\_t entry\_count; /\* количество дескрипторов в таблице дескрипторов \*/

byte4\_t data\_block\_start; /\* смещение к началу данных файловой системы на блочном устройстве \*/

byte4\_t reserved[MYRWFS\_BLOCK\_SIZE\_BYTES / 4 - 8];

} myrwfs\_super\_block\_t;

Файловый дескриптор, расположенный на диске:

typedef struct myrwfs\_file\_entry

{

char name[MYRWFS\_FILENAME\_LEN + 1];

byte4\_t size; /\* размер дескриптора файла в байтах \*/

byte4\_t perms; /\* разрешения \*/

byte4\_t blocks[MYRWFS\_DATA\_BLOCK\_CNT];

} myrwfs\_file\_entry\_t;

Определим структуру, которая ставит в соответствие стуктуру суперблок на диске и структуру суперблок пространства ядра:

typedef struct myrwfs\_info

{

struct super\_block \*vfs\_sb; /\* обратный указатель на структуру суперблок пространства ядра \*/

myrwfs\_super\_block\_t myrwfs\_sb; /\* указатель на структуру суперблок, расположенную на устройстве \*/

byte1\_t \*used\_blocks; /\* указатель на количество блоков, помеченных как занятые \*/

} myrwfs\_info\_t;

Обратим внимание! Поле byte1\_t \*used\_blocks; содержит массив флагов (значения 0 или 1), показывающих, занят ли соответствующий блок блочного устройства. При записи данных на блок, его необходимо пометить, как занятый значением 1. При освобождении блока - пометить как свободный значением флага 0.

Кроме того, определим функции, ставящие в соответствие файловые дескрипторы на диске и пространства ядра. Отметим также, что файловые дескриптор корневого каталога файловой системы, располагается в таблице дескрипторов с индекса 1, так как нулевой inode зарезервирован для представления невалидного указателя.

#define ROOT\_INODE\_NUM (1)

#define MYRWFS\_TO\_VFS\_INODE\_NUM(i) (ROOT\_INODE\_NUM + 1 + (i))

#define VFS\_TO\_MYRWFS\_INODE\_NUM(i) ((i) - (ROOT\_INODE\_NUM + 1))

Введем также следующее соотношение: таблица файловых дескрипторов занимает 10% размера блочного устройства.

// источник: https://sysplay.in/blog/linux-device-drivers/2014/07/file-systems-the-semester-project/

#define MYRWFS\_ENTRY\_FRACTION 0.10 /\* 10% блоков резервируем под таблицу дескрипторов - эмпирически \*/

Поскольку структура суперблок расположена в первом блоке блочного устройства, смещение к началу таблицы дескрипторов определим равным единице.

#define MYRWFS\_ENTRY\_TABLE\_BLOCK\_START 1

Для форматирования блочного устройства реализуем функции записи на него структуры суперблок и таблицы файловых дескрипторов:

void write\_super\_block(int fd, myrwfs\_super\_block\_t \*myrwfs\_sb) {

write(fd, myrwfs\_sb, sizeof(myrwfs\_super\_block\_t));

}

void write\_file\_entries\_table(int fd, myrwfs\_super\_block\_t \*myrwfs\_sb) {

byte1\_t block[MYRWFS\_BLOCK\_SIZE\_BYTES];

for (int i = 0; i < myrwfs\_sb->block\_size\_bytes / myrwfs\_sb->entry\_size\_bytes; i++) {

memcpy(block + i \* myrwfs\_sb->entry\_size\_bytes, &fe, sizeof(fe));

}

for (int i = 0; i < myrwfs\_sb->entry\_table\_size\_blocks; i++) {

write(fd, block, sizeof(block));

}

}

? что в этой функции происходит и что такое fe. РАЗМЕР БЛОКА / РАЗМЕР ДЕСКРИПТОРА ФАЙЛА. То есть мы заполняем подряд с помощью fe

После выполнения форматирования блочного устройства, может быть выполнено монтирование файловой системы.

**Реализация файловой системы.**

Файловая система реализована в виде загружаемого модуля ядра.

Рассмотрим функции инициализации, чтения и записи данных, расположенных на отформатированном блочном устройстве. Эти функции будут вызваны в реализациях вызовов VFS высокого уровня.

Для понимания работы функций предварительно рассмотрим вызовы чтения и записи данных на блочное устройство. Операционные системы Unix/Linux предоставляют системные вызовы sb\_bread, brelse и mark\_buffer\_dirty для работы с блочными устройствами. Вспомним, что блочное устройство является буферизованным - то есть запись данных пользователем выполняется в буфер, после чего операционная система неявно для пользователя копирует данные из буфера на устройство.

sb\_bread: <https://elixir.bootlin.com/linux/latest/ident/sb_bread>

* reads a block with the given number in a buffer\_head structure; in case of success returns a pointer to the buffer\_head structure, otherwise it returns NULL; the size of the read block is taken from the superblock, as well as the device from which the read is done;
* Если необходимо прочитать данные с блочного устройства, не передавая информацию о файловой системе, под которую оно было отформатировано, необходимо воспользоваться аналогичной функцией bread.

brealse: <https://www.unix.com/man-page/netbsd/9/brelse/>

* Unbusy a buffer and release it to the free lists.

mark\_buffer\_dirty: [https://manpages.debian.org/experimental/linux-manual-4.12/mark\_buffer\_dirty.9.en.html#SYNOPSIS](https://manpages.debian.org/experimental/linux-manual-4.12/mark_buffer_dirty.9.en.html" \l "SYNOPSIS)

* mark a buffer\_head as needing writeout

Рассмотрим реализации функций чтения и записи на блочное устройство.

 struct block\_device      \*s\_bdev;          /\* соответствующий драйвер блочного устройства \*/

static int read\_sb\_from\_myrwfs(myrwfs\_info\_t \*info, myrwfs\_super\_block\_t \*myrwfs\_sb) {

struct buffer\_head \*bh;

if (!(bh = sb\_bread(info->vfs\_sb, 0))) { //0, тк sb там

return -1;

}

memcpy(myrwfs\_sb, bh->b\_data, MYRWFS\_BLOCK\_SIZE\_BYTES);

brelse(bh);

return 0;

}

static int read\_entry\_from\_myrwfs(myrwfs\_info\_t \*info, int ino, myrwfs\_file\_entry\_t \*fe) {

byte4\_t offset = ino \* sizeof(myrwfs\_file\_entry\_t);

byte4\_t len = sizeof(myrwfs\_file\_entry\_t);

byte4\_t block = info->myrwfs\_sb.entry\_table\_block\_start;

byte4\_t block\_size\_bytes = info->myrwfs\_sb.block\_size\_bytes;

byte4\_t bd\_block\_size = info->vfs\_sb->s\_bdev->bd\_block\_size;

byte4\_t abs;

struct buffer\_head \*bh;

abs = block \* block\_size\_bytes + offset;

block = abs / bd\_block\_size;

offset = abs % bd\_block\_size;

if (offset + len > bd\_block\_size) {

return -1;

}

if (!(bh = sb\_bread(info->vfs\_sb, block))) {

return -1;

}

memcpy((void \*) fe, bh->b\_data + offset, len);

brelse(bh);

return 0;

}

static int write\_entry\_to\_myrwfs(myrwfs\_info\_t \*info, int ino, myrwfs\_file\_entry\_t \*fe) {

byte4\_t offset = ino \* sizeof(myrwfs\_file\_entry\_t);

byte4\_t len = sizeof(myrwfs\_file\_entry\_t);

byte4\_t block = info->myrwfs\_sb.entry\_table\_block\_start;

byte4\_t block\_size\_bytes = info->myrwfs\_sb.block\_size\_bytes;

byte4\_t bd\_block\_size = info->vfs\_sb->s\_bdev->bd\_block\_size;

struct buffer\_head \*bh;

byte4\_t abs = block \* block\_size\_bytes + offset;

block = abs / bd\_block\_size;

offset = abs % bd\_block\_size;

if (offset + len > bd\_block\_size) {

return -1;

}

if (!(bh = sb\_bread(info->vfs\_sb, block))) {

return -1;

}

memcpy(bh->b\_data + offset, (void \*) fe, len);

mark\_buffer\_dirty(bh);

brelse(bh);

return 0;

}

Обратим особое внимание на следующие преобразования:

byte4\_t offset = ino \* sizeof(myrwfs\_file\_entry\_t);

byte4\_t len = sizeof(myrwfs\_file\_entry\_t);

byte4\_t block = info->myrwfs\_sb.entry\_table\_block\_start;

byte4\_t block\_size\_bytes = info->myrwfs\_sb.block\_size\_bytes;

byte4\_t bd\_block\_size = info->vfs\_sb->s\_bdev->bd\_block\_size;

struct buffer\_head \*bh;

byte4\_t abs = block \* block\_size\_bytes + offset;

block = abs / bd\_block\_size;

offset = abs % bd\_block\_size;

if (offset + len > bd\_block\_size) {

return -1;

}

Функции принимают на вход индекс файлового дескриптора в таблице файловых дескрипторов. Для адресации данных на блочном устройстве, этому индексу должно быть поставлено в соответствие смещение к блоку, на котором расположен файловый дескриптор.

С учетом реализаций функций чтения и записи файловых дескрипторов на блочное устройство, функции создания и поиска файлового дескриптора будут иметь следующий вид:

int myrwfs\_create(myrwfs\_info\_t \*info, char \*fn, int perms, myrwfs\_file\_entry\_t \*fe) {

int ino, free\_ino, i;

free\_ino = -1;

for (ino = 0; ino < info->myrwfs\_sb.entry\_count; ino++) {

if (read\_entry\_from\_myrwfs(info, ino, fe) < 0)

return -1;

if (!fe->name[0]) {

free\_ino = ino;

break;

}

}

if (free\_ino == -1) {

printk(KERN\_ERR "No entries left\n");

return -1;

}

strncpy(fe->name, fn, MYRWFS\_FILENAME\_LEN);

fe->name[MYRWFS\_FILENAME\_LEN] = 0;

fe->size = 0;

fe->perms = perms;

for (i = 0; i < MYRWFS\_DATA\_BLOCK\_CNT; i++) {

fe->blocks[i] = 0;

}

if (write\_entry\_to\_myrwfs(info, free\_ino, fe) < 0)

return -1;

return MYRWFS\_TO\_VFS\_INODE\_NUM(free\_ino);

}

int myrwfs\_lookup(myrwfs\_info\_t \*info, char \*fn, myrwfs\_file\_entry\_t \*fe) {

for (int ino = 0; ino < info->myrwfs\_sb.entry\_count; ino++) {

if (read\_entry\_from\_myrwfs(info, ino, fe) < 0)

return -1;

if (!fe->name[0])

continue;

if (strcmp(fe->name, fn) == 0)

return MYRWFS\_TO\_VFS\_INODE\_NUM(ino);

}

return -1;

}

При реализации файловой системы для семейства Unix\Linux необходимо реализовать функции инициализации и уничтожения структуры суперблок пространства ядра. Однако в случае монтируемой файловой системы, необходимо кроме того определить функции инициализации и уничтожения структуры суперблок, расположенной на диске:

int fill\_myrwfs\_info(myrwfs\_info\_t \*info) {

if (read\_sb\_from\_myrwfs(info, &info->myrwfs\_sb) < 0) {

return -1;

}

if (info->myrwfs\_sb.magic\_number != MYRWFS\_MAGIC\_NUMBER) {

printk(KERN\_ERR "Invalid MYRWFS detected. Giving up.\n");

return -1;

}

byte1\_t \*used\_blocks = (byte1\_t \*)(vmalloc(info->myrwfs\_sb.bdev\_size\_blocks));

if (!used\_blocks) {

return -ENOMEM;

}

int i;

for (i = 0; i < info->myrwfs\_sb.data\_block\_start; i++) {

used\_blocks[i] = 1;

}

for (; i < info->myrwfs\_sb.bdev\_size\_blocks; i++) {

used\_blocks[i] = 0;

}

myrwfs\_file\_entry\_t fe;

for (int i = 0; i < info->myrwfs\_sb.entry\_count; i++) {

if (read\_entry\_from\_myrwfs(info, i, &fe) < 0) {

vfree(used\_blocks);

return -1;

}

if (!fe.name[0])

continue;

for (int j = 0; j < MYRWFS\_DATA\_BLOCK\_CNT; j++) {

if (fe.blocks[j] == 0) break;

used\_blocks[fe.blocks[j]] = 1;

}

}

info->used\_blocks = used\_blocks;

info->vfs\_sb->s\_fs\_info = info;

return 0;

}

void kill\_mywfs\_info(myrwfs\_info\_t \*info) {

if (info->used\_blocks)

vfree(info->used\_blocks);

}

Обратим особое внимание на участок кода:

...

byte1\_t \*used\_blocks = (byte1\_t \*)(vmalloc(info->myrwfs\_sb.bdev\_size\_blocks));

if (!used\_blocks) {

return -ENOMEM;

}

int i;

for (i = 0; i < info->myrwfs\_sb.data\_block\_start; i++) {

used\_blocks[i] = 1;

}

for (; i < info->myrwfs\_sb.bdev\_size\_blocks; i++) {

used\_blocks[i] = 0;

}

...

Вспомним, что для записи данных в i-й блок блочного устройства, его необходимо пометить как занятый, установив значение флага used\_blocks[i] в 1. С учетом этой информации, реализуем функции аллокации и освобождения блока:

int myrwfs\_get\_data\_block(myrwfs\_info\_t \*info) {

int i;

for (i = info->myrwfs\_sb.data\_block\_start; i < info->myrwfs\_sb.bdev\_size\_blocks; i++) {

if (info->used\_blocks[i] == 0) {

info->used\_blocks[i] = 1;

return i;

}

}

return -1;

}

void myrwfs\_put\_data\_block(myrwfs\_info\_t \*info, int i) {

info->used\_blocks[i] = 0;

}

Теперь мы можем реализовать функции высокого уровня для работы с нашей файловой системой. Реализации этих функций будут вызваны операционной системой при выполнении вызовов VFS - ls, cat, touch и т.д.

В функции инициализации структуры суперблок пространства ядра необходимо вызвать также функцию инициализации стурктуры суперблок, расположенной на диске:

static int myrwfs\_fill\_super(struct super\_block \*sb, void \*data, int silent) {

printk(KERN\_INFO "\*\* MYRWFS: myrwfs\_fill\_super\n");

myrwfs\_info\_t \*info;

if (!(info = (myrwfs\_info\_t \*)(kzalloc(sizeof(myrwfs\_info\_t), GFP\_KERNEL))))

return -ENOMEM;

info->vfs\_sb = sb;

if (fill\_myrwfs\_info(info) < 0) {

kfree(info);

return -1;

}

sb->s\_magic = info->myrwfs\_sb.magic\_number;

sb->s\_blocksize = info->myrwfs\_sb.block\_size\_bytes;

sb->s\_blocksize\_bits = log\_base\_2(info->myrwfs\_sb.block\_size\_bytes);

sb->s\_type = &myrwfs;

sb->s\_op = &myrwfs\_sops;

myrwfs\_root\_inode = iget\_locked(sb, ROOT\_INODE\_NUM);

if (!myrwfs\_root\_inode) {

kill\_mywfs\_info(info);

kfree(info);

return -1;

}

if (myrwfs\_root\_inode->i\_state & I\_NEW) {

myrwfs\_root\_inode->i\_op = &myrwfs\_iops;

myrwfs\_root\_inode->i\_mode = S\_IFDIR | S\_IRWXU | S\_IRWXG | S\_IRWXO;

myrwfs\_root\_inode->i\_fop = &myrwfs\_dops;

unlock\_new\_inode(myrwfs\_root\_inode);

}

sb->s\_root = d\_make\_root(myrwfs\_root\_inode);

if (!sb->s\_root) {

iget\_failed(myrwfs\_root\_inode);

kill\_mywfs\_info(info);

kfree(info);

return -ENOMEM;

}

return 0;

}

Аналогично, в функции, выполняющей уничтожение структуры суперблок пространства ядра, необходимо также выполнить освобождение дескрипторов структур, расположенных на диске.

static void myrwfs\_put\_super(struct super\_block \*sb) {

myrwfs\_info\_t \*info = (myrwfs\_info\_t \*)(sb->s\_fs\_info);

printk(KERN\_INFO "\*\* MYRWFS: myrwfs\_put\_super\n");

if (info) {

kill\_mywfs\_info(info);

kfree(info);

sb->s\_fs\_info = NULL;

}

}

Обратим внимание также на то, что в функции примонтирования файловой системы, необходимо вызвать mount\_bdev и указать флаг FS\_REQUIRES\_DEV в дескрипторе файловой системы:

static struct dentry \*myrwfs\_mount(struct file\_system\_type \*fs, int flags, const char \*devname, void \*data) {

printk(KERN\_INFO "\*\* MYRWFS: myrwfs\_mount: devname = %s\n", devname);

return mount\_bdev(fs, flags, devname, data, &myrwfs\_fill\_super);

}

static struct file\_system\_type myrwfs = {

name: "myrwfs",

fs\_flags: FS\_REQUIRES\_DEV,

mount: myrwfs\_mount,

kill\_sb: kill\_block\_super,

owner: THIS\_MODULE

};

Функции создания и поиска файлового дескриптора в таблице файловых дескрипторов будут иметь следующий вид - отметим, что в реализации этих функций вызываются функции низкого уровня, выполняющие чтение и запись данных, расположенных на блочном устройстве.

static int myrwfs\_inode\_create(struct inode \*parent\_inode, struct dentry \*dentry, umode\_t mode, bool excl) {

char fn[dentry->d\_name.len + 1];

int perms = 0;

myrwfs\_info\_t \*info = (myrwfs\_info\_t \*)(parent\_inode->i\_sb->s\_fs\_info);

int ino;

struct inode \*file\_inode;

myrwfs\_file\_entry\_t fe;

printk(KERN\_INFO "\*\* MYRWFS: myrwfs\_inode\_create\n");

strncpy(fn, dentry->d\_name.name, dentry->d\_name.len);

fn[dentry->d\_name.len] = 0;

if (mode & (S\_IRUSR | S\_IRGRP | S\_IROTH))

mode |= (S\_IRUSR | S\_IRGRP | S\_IROTH);

if (mode & (S\_IWUSR | S\_IWGRP | S\_IWOTH))

mode |= (S\_IWUSR | S\_IWGRP | S\_IWOTH);

if (mode & (S\_IXUSR | S\_IXGRP | S\_IXOTH))

mode |= (S\_IXUSR | S\_IXGRP | S\_IXOTH);

perms |= (mode & S\_IRUSR) ? 4 : 0;

perms |= (mode & S\_IWUSR) ? 2 : 0;

perms |= (mode & S\_IXUSR) ? 1 : 0;

if ((ino = myrwfs\_create(info, fn, perms, &fe)) == -1)

return -1;

file\_inode = new\_inode(parent\_inode->i\_sb);

if (!file\_inode)

{

myrwfs\_remove(info, fn);

return -ENOMEM;

}

printk(KERN\_INFO "\*\* MYRWFS: Created new VFS inode for #%d, let's fill in\n", ino);

file\_inode->i\_ino = ino;

file\_inode->i\_size = fe.size;

file\_inode->i\_mode = S\_IFREG | mode;

file\_inode->i\_fop = &myrwfs\_fops;

if (insert\_inode\_locked(file\_inode) < 0)

{

make\_bad\_inode(file\_inode);

iput(file\_inode);

myrwfs\_remove(info, fn);

return -1;

}

d\_instantiate(dentry, file\_inode);

unlock\_new\_inode(file\_inode);

return 0;

}

static struct dentry \*myrwfs\_inode\_lookup(struct inode \*parent\_inode, struct dentry \*dentry, unsigned int flags) {

myrwfs\_info\_t \*info = (myrwfs\_info\_t \*)(parent\_inode->i\_sb->s\_fs\_info);

char fn[dentry->d\_name.len + 1];

int ino;

myrwfs\_file\_entry\_t fe;

struct inode \*file\_inode = NULL;

printk(KERN\_INFO "\*\* MYRWFS: myrwfs\_inode\_lookup\n");

if (parent\_inode->i\_ino != myrwfs\_root\_inode->i\_ino)

return ERR\_PTR(-ENOENT);z

strncpy(fn, dentry->d\_name.name, dentry->d\_name.len);

fn[dentry->d\_name.len] = 0;

if ((ino = myrwfs\_lookup(info, fn, &fe)) == -1)

return d\_splice\_alias(file\_inode, dentry);

file\_inode = iget\_locked(parent\_inode->i\_sb, ino);

if (!file\_inode)

return ERR\_PTR(-EACCES);

if (file\_inode->i\_state & I\_NEW) {

file\_inode->i\_size = fe.size;

file\_inode->i\_mode = S\_IFREG;

file\_inode->i\_mode |= ((fe.perms & 4) ? S\_IRUSR | S\_IRGRP | S\_IROTH : 0);

file\_inode->i\_mode |= ((fe.perms & 2) ? S\_IWUSR | S\_IWGRP | S\_IWOTH : 0);

file\_inode->i\_mode |= ((fe.perms & 1) ? S\_IXUSR | S\_IXGRP | S\_IXOTH : 0);

file\_inode->i\_fop = &myrwfs\_fops;

unlock\_new\_inode(file\_inode);

}

d\_add(dentry, file\_inode);

return NULL;

}

static struct inode\_operations myrwfs\_iops = {

create: myrwfs\_inode\_create,

lookup: myrwfs\_inode\_lookup

};